TABLE OF CONTENTS

ASSEMBLER

INTRODUCTION

This section is an overview of the 2500 A.D. 8748 Cross Assembler. The intent of this manual is to describe the operation of the Cross Assembler. It is assumed that the user is familiar with the 8748 operation and instruction set.

The 2500 A.D. 8748 Cross Assembler enables the user to write programs on a 8086 based system, which can then be assembled into relocatable object code and linked to the desired execution address using the 2500 A.D. Linker. Two different syntaxes are accepted. The first is the standard 8748 syntax as defined by Intel, and the second is a form similar to Zilog's Z80 syntax. For users familiar with this syntax, this helps to speed up the switch from one processor to another. The mnemonics used for either syntax are those defined by Intel.

The Assembler will process files as large as the disk storage device. All buffers including the Symbol Table Buffer overflow to disk. The Macro section handles string comparisons, dummy parameter substitution, recursion and nesting limited only by the amount of disk storage space available.

The Conditional Assembly section enables the user to direct the Assembler to process different sections of the source file depending on the outcome of assembly time operations. Conditionals may be nested to 248 levels, and the Assembler aids the programmer in detecting conditional nesting errors by not only checking for unbalanced conditional levels, but also by displaying the current active conditional level in the object code field of the listing.

The Assembly Time Calculation section will perform calculations with up to 16 pending operands, using either 16 or 32 bit arithmetic. The algebraic hierarchy may be changed through the use of parenthesis.

The Listing Control section provides for listing all or just sections of the program, with convenient Assembler error detection overrides, along with Assembly Run Time Commands that may be used to dynamically change the listing mode.

The 2500 A.D. Linker allows up to 128 files to either be linked together or just used for external reference resolution. As with the Assembler, all buffers used by the Linker overflow to disk, with the result being that Globals, Externals and object code sizes are limited only by the amount of disk storage space available. In an effort to accommodate different output requirements, multiple versions of the Linker are provided. See the Linker section of this manual for a description of the different Linkers.

8748 CROSS ASSEMBLER OPERATING INSTRUCTIONS

PROMPT MODE

To run the Assembler type : X8748

The Assembler will respond with :

LISTING DESTINATION ? (TI,LP,DD,NL,DL,EO)

with the abbreviations as follows:

TI = Terminal Immediate
LP = Line Printer
DD = Disk Drive
NL = No Listing
DL = Directive Listing
EO = Error Only

If the listing is to be sent to the disk, the Assembler will output the following prompt:

LISTING DRIVE ? (<CR> = SAME AS OUTPUT FILE)

The user may enter a drive letter or just a carriage return, which will send the listing to the same drive that the output file is going to be sent to. The listing file always receives the same name as the input file with an extension of LST. Also, when the listing is being sent to the disk, an additional tab is inserted between the end of the object code field and the start of the label field so that when the file is printed, the tabs will be expanded properly.

After this the Assembler prompts the operator for the source code filename as shown below.

INPUT FILENAME ? :

Finally, the Assembler asks for the output filename.

OUTPUT FILENAME ? :

If the user repsonds to the above prompt with just a carriage return, the output file will receive the same filename as the input file, with an extension of OBJ. If the response is just a drive letter followed by a colon, the output will be sent to that drive with the same filename as the input file, with an extension of OBJ. If the repsonse is a filename with no extension, the output file will be under that filename with an extension of OBJ.

If the listing is to be under assembler directive control the additional prompt shown below is output:

ASSEMBLER DIRECTIVE LISTING DESTINATION ? (TI,LP,DD)

The directive control listing allows the user to output only the part of the assembly specified by LIST ON/OFF. For more information on LIST ON/OFF see the listing control section.

Cross Reference generates a listing that allows the user to locate symbols used anywhere in the program.

If Error Only is chosen the following destinations will be shown.

ASSEMBLER DIRECTIVE LSITING DESTINATION ? (TI,LP,DD)

COMMAND LINE MODE

The Assembler may also be invoked using a command line format. This mode is useful for Submit type processing. The only difference in the operation of the command line mode as compared with the prompt mode is that the responses to the prompts are included in the command line, with each response separated by either spaces or a comma as shown below.

```
X8748 listing_destination,input_filename,output_filename
```

For example, if the input filename is TEST.SRC, the output filename is TEST.OBJ and no listing is desired the command line would be:

```
X8748  NL   TEST.SRC   TEST.OBJ
```

or

```
X8748  NL,TEST.SRC,TEST.OBJ
```

If the file contains conditional listing directives, and the specified sections are to be sent to the line printer, and cross reference is not specified then the following command line would be issued (assuming the same filenames as above) :

```
X8748    DL   LP  N  TEST.SRC   TEST.OBJ
```

If it was desired to send the listing in the above example to the disk instead of to the printer, the following command line would be issued:

```
X8748    DL   DD   A  N TEST.SRC    TEST.OBJ
```

Note that the listing drive number is included in the command line. If you want the listing to go to the same drive as the output file, then a command line using commas as place holders must be used as shown below.

```
X8748    DL,DD,,N,TEST.SRC,TEST.OBJ
```

If the ".ASK" assembler directive is used, the responses must be appended in the proper order at the end of the command line. For example, if in the last example the file contained two ".ASK" directives which would be repsonded to with a 0 and a 1 then the following command line would be issued:

```
X8748    DL   LP   TEST.SRC   TEST.OBJ   0   1
```

The command line buffer is 128 bytes long and no checks are made for overflow. Also, note that the Assembler Run Time Listing Commands interrogate only the console and cannot be controlled through a Submit file. However, they are active during Submit file processing.

SYSTEM DEFAULTS

The following default filename extensions will be used by the 2500 A.D. programs if no extension is specified by the user.

ASM - Input to the Assembler
OBJ - Output from the Assembler
LST - Listing file

OBJ - Input to the Linker
TSK - Output from LINK1
HEX - Output from LINK2

Note that because of the additional information included in the Assembler output file, the Linker must always be run, even if the program is assembled at the desired run address and there are no external references. This is just so that all the additional information can be removed and a load file can be generated.

ASSEMBLER RUN TIME COMMANDS

The following commands are active during the assembly process. Before parsing each line, the console is checked for input. These commands are active during pass 1 as well as pass 2, and override the listing mode specified when the Assembler was first activated. The ^ symbol denotes pressing the control key simultaneously with the letter key.

^S = Alternately start and stop the assembly
^Z = Terminate the assembly
^T = Display the output at the terminal
^P = Display the output at the printer
^D = Send the output to the disk
^B = Both terminal & printer or disk
^N = Turn off the output display

When the ^B selection is used, if the listing is currently being displayed at the terminal, the printer is included. If the listing is currently being sent to the printer, the terminal is included. If the listing is being sent to the disk, the terminal is included. If the ^D function is used, and the disk file has not been created yet, the Assembler will create it before it continues with the assembly.

ASSEMBLY ERROR PROCESSING

When an assembly error is encountered, the action taken by the Assembler depends on the listing mode it is currentiy operating under.

If the No List option was specified, the statement causing the error and the error message will be output to the terminal, the display will be turned on and the Assembler will halt just as if the user had typed ^S. The reason for this is to give the user a chance to see exactly where the error is. This will occur on pass 1 as well as pass 2. Note that some errors are not detectable on pass 1, such as undefined symbols. After the error has been displayed, the output can be turned off using ^N.

If the listing is being sent to the printer, then errors encountered on pass 1 are sent to the terminal but not the printer, and the Assembler does not halt. On pass 2, the error is output to the printer as well as the terminal and the assembiy continues.

If the listing is being sent to the printer under assembler directive control, any errors encountered during pass 1 are output to the terminal but not the printer, and the assembiy continues. Errors detected during pass 2 are output to the printer and the terminal, even if the error is not inside a block that was specified to be listed.

Because of the action taken by the Assembler outside of LIST ON/OFF blocks when Directive Listing is specified, error only outputs can be obtained by removing all LIST ON/OFF blocks and specifying Directive Listing (DL) for output destination.

A Symbol Table is output when the listing destination is either the disk or the printer. The Symbol Table is not output to the terminal unless the output is being sent to both the terminal and the printer or the terminal and the disk. If the output is under Assembler Directive control, the Symbol Table is output if a LIST ON block is being processed when the end of the program is reached. Otherwise, the Symbol Table is not output.

ASSEMBLER FILE HANDLING DESCRIPTION

During the Assembly process the Assembler manipulates the following files:

Source file	- The file containing the program source code.
Object file	- The output file from the assembler containing object code, address relocation information, External and Global symbols with pertinate address information and program entry point information.
Relocation file	- This is a temporary file used to hold the address relocation information during the assembly.
Symbol Table file	- This file is used when the symbol table overflows. The table overflows at about 550 symbols.
Macro Table file	- This file is used to store the Macro names when the Macro table overflows. The table overflows after 204 Macros have been defined.
Macro Buffer file	- This file is used to store the Macro definitions and dummy parameter lists. The parameter list may be up to 80 characters. Also, this file holds information for macro nesting, which is limited only by the size of the disk.
External Symbol file	- This file holds information concerning the address of External symbols used in the program.

With the exception of the Source file, all the files are stored on the same drive as the Object file. Since every buffer used by the assembler overflows to disk, the only limit to the size of the program that can be assembled is the amount of disk space available.

At the end of Pass 1, the Assembler searches the Symbol Table for multiply defined symbols. For programs with less than about 600 symbols, this search time is not noticable. For programs at or above this threshold the user may notice a period of a few seconds where it appears that the Assembler quit running. The purpose of this notice is to prevent the user from having heart failure over this event.

8748 ASSEMBLY LANGUAGE SYNTAX

This section describes the syntax used by the 2500 A.D. Cross Assembler.

NUMBER BASE DESIGNATIONS

Number bases are specified by the following:

Binary	-	B
Octal	-	Q
Decimal	-	D or no base designation
HEX	-	H
ASCII	-	"X" or 'X'

ASCII special cases for program clarity:

"CR" or 'CR'	-	Carriage return
"LF" or 'LF'	-	Line feed
"SP" or 'SP'	-	Space
"HT" or 'HT'	-	Horizontal tab
"NL" or 'NL'	-	Null

PROGRAM COMMENTS

Comment lines must start with a semi-colon in column 1, unless the .'COMMENT' directive is used. Comments after an instruction do not need a semi-colon if at least 1 space or tab preceeds the start of the comment.

LABELS

Labels may be any number of characters long, but only 10 characters are significant. Labels may start in any column if the name is terminated by a colon. If no colon is used, the label must start in column 1. All labels must start with an alpha character. Upper and lower case characters are considered to be different.

PROGRAM COUNTER ($)

The special character '$' may be used in an expression to specify the program counter. The value assigned to the dollar sign is the program counter value at the start of the instruction.

UPPER/LOWER CASE

All mnemonics and register specifications must use upper case characters. Lower and upper case characters in labels are considered to be different. Hexadecimal characters must be upper case. Included with this package is a program called 'CASE.COM' to convert source code from lower to upper case. In contrast with most general purpose routines that perform this function, Case's claim to fame is that it will leave Ascii strings bracketed with apostrophes alone. Note however, that it will change all label characters to upper case. See the Converter Operating Instructions section of this manual for the use of this utility.

ADDRESSING MODES

The 8748 addressing modes are listed below, along with examples of each type. The Cross Assembler will accept either of the two different syntaxes shown for Immediate and Register Indirect mode.

IMMEDIATE

The data is contained in the instruction.

Examples:

```
MOV     A,#12H          ; Load A with the Hex Number 12
MOV     A,#12H          ; Same as above with optional # sign
MOV     A,#DATA         ; Ld A with the value associated
                          with the label DATA
```

REGISTER

The data is contained in a CPU register.

Examples:

```
MOV     R0,A            ; Ld the contents in register A into
                          register R0
INC     R0              ; Increment the contents of register
                          R0 by one
```

REGISTER INDIRECT

The address of the operand is pointed to by the contents of a register.

Examples:

```
MOV     A,@R0           ; Load A with the contents of the
                          memory location pointed to by R0
MOV     A,(R0)          ; Same instruction as above, with
                          the second Register Indirect form
```

DIRECT ADDRESS

The address of the operand is contained in the instruction.

Examples:

```
JMP     ADDRESS         ; Jump to the location 'ADDRESS'
```

ASSEMBLER DIRECTIVES

This section describes the Assembler Directives. Directives may be preceeded by a decimal point if desired to help differentiate them from program instructions.

STORAGE CONTROL

```
ORG       EXP
ORIGIN
```

Sets the program assembly address. If this directive is not executed, the assembly address defaults to 0000.

```
END
```

Defines the end of the program. Every program must have an END statement, and every program INCLUDED in the assembly must have an END statement.

```
LABEL:    DB        EXP
          DEFB
          BYTE
          STRING
```

The Assembler will store the value of the expression in consecutive memory locations. The BYTE expression may be any mixture of operand types with each one spearated by a comma. Ascii character strings must be bracketed by apostrophes. If the string contains an apostrophe, this can be specified with two apostrophes in a row. If no expression is given, one byte is reserved and zeroed. A label is optional.

```
.BYTE                    ;Reserves 1 zeroed byte.
.BYTE     10             ;Reserves 1 byte = 10 decimal.
.BYTE     1,2,3          ;Reserves 3 bytes, = to 1,2 &
                          3 in that order.
.BYTE     SYMBOL-10      ;Searches the symbol table for
                          SYMBOL, subtracts 10 decimal
                          from it's value, and stores
                          the result.
.BYTE     'Hello'        ;Stores the Ascii equivalent
                          of the string Hello in con-
                          secutive memory locations.
.BYTE     'Hello', 0DH   ;Same as above example, with
                          the addition of a carriage
                          return at the end. Spaces are
                          ignored before operands, but
                          the comma is required.
.BYTE     '2500 A.D.''s' ;Embedded apostrophe.
```

```
LABEL:      DW          EXP
            DEFW
            WORD
```

Operates exactly as BYTE except a full word is used for each operand. The value is stored most significant byte first. A label is optional.

```
LABEL:      LWORD       EXP
            LONGW
```

Operates exactly as BYTE except a long word (32 bits) is used for each operand. The value is stored as follows:

```
BYTE 0 = BITS  0 -  7
BYTE 1 = BITS  8 - 15
BYTE 2 = BITS 16 - 23
BYTE 3 = BITS 24 - 31
```

A label is optional.

```
LABEL:      ASCII       STRING
```

Stores STRING in memory up to but not including either a carriage return or a ";" (HEX 7C). A label is optional. Following are some examples of ASCII.

```
            ASCII       Hello           ;Stores the Ascii representa-
                                         tion of Hello in consecutive
                                         memory locations. Incidental-
                                         ly, this comment would be
                                         stored also.
            ASCII       Hello;          ;Now the comment wouldn't be
                                         stored.  The next example
                                         shows termination with just a
                                         carriage return.
            ASCII       Hello
```

```
LABEL:      DS          EXP
            DEFS
            BLKB
```

Reserves the number of bytes specified by 'EXP'. The reserved bytes are set equal to 0. A label is optional.

```
LABEL:      BLKW        EXP
```

Reserves the number of words specified by 'EXP'. The reserved words are zeroed. A label is optional.

DEFINITION CONTROL

LABEL: EQU EXP
EQUAL

Equates 'LABEL' to the value of 'EXP'.

LABEL: VAR EXP
DEFL

Equates 'LABEL' to 'EXP' but may be changed as often as desired throughout the program. A label defined as a variable should not be redefined by an 'EQUAL' directive.

LABEL: MACRO ARGS

Specifies the start of a Macro Definition.

ENDM
MACEND

Specifies the end of a Macro Definition.

MACEXIT

This directive causes the immediate exit from a macro. The difference between MACEXIT and MACEND is that during the macro definition process, MACEXIT does not terminate the macro, and if MACEXIT is in the path of a false conditional assembly block, it is not executed.

EXTERN LABEL
EXTERNAL

Specifies the label as being defined in another program. Multiple labels may be specified as long as each one is separated by a comma.

GLOBAL LABEL
PUBLIC

Specifies the label as a global label that may be referenced by other programs. Multiple labels may be specified as long as each one is separated by a comma. Below are some examples of the correct use of GLOBAL.

```
        GLOBAL    SYM1          ;Declares the label SYM1 to be
                                  accessible to other programs.
                                  The Linker will resolve the
                                  external references.
        GLOBAL    SYM1, SYM2    ;Multiple declarations on the
                                  same line are legal if sepa-
                                  rated by a comma. The spaces
                                  are ignored.
```

LABEL: **ASK** PROMPT

Outputs 'PROMPT' to the terminal and waits for a 1 character response, from which 30 hex is subtracted. The purpose of this is usually to introduce a 0/1 flag into the program. 'LABEL' is set equal to the result. A carriage return terminates 'PROMPT'. On pass 2, the line is output along with the response.

The following is an example of 'ASK' :

```
DISK_SIZE:  ASK  ASSEMBLE FOR  8" (=1) OR 5 1/4" (=0) DRIVES ? :
```

ASSEMBLY MODE

RADIX EXP

Sets the Assembler number base as follows:

2 or B = Binary
8 or Q = Octal
10 or D = Decimal
16 or H = Hexadecimal

No expression = return to default mode which is base 10, and assume all others will be designated with B, Q, D or H after the constant.

CODE

Specifies the start of a program instruction section. This directive can be used in conjunction with the DATA directive to generate separate Code and Data program sections. The Assembler initially defaults to a CODE section. See the Linker Operating Instructions section of this manual for information on the different ways the Linker can be directed to process Code and Data sections.

DATA

Specifies the start of a program data section. This directive can be used in conjunction with the CODE directive to generate separate Code and Data program sections. The Assembler initially defaults to a CODE section. See the Linker Operating Instructions section of this manual for information on the different ways the Linker can be directed to process Code and Data sections.

MOD32 ON

This directive specifies the precision of intermediate assembly time calculations. The Assembler defaults to 16 bit calculations. Executing this directive causes the Assembler to perform all calculations using 32 bits. This directive is useful when intermediate calculation results may be larger than 16 bits, and overflow would result. The final result however, is a 16 bit result and is stored as such. See the Assembly Time Calculations section of this manual for a further discussion of these two different calculation modes.

MOD32 OFF

This directive returns the Assembler to the default mode of using only 16 bits for all assembly time calculations. See the Assembly Time Calculations section of this manual for a further discussion of the two different calculation modes.

COMMENT X

Directs the Assembler to consider all lines starting with this one until the next 'X' to be comments. 'X' can be any character.

INCLUDE filename

Directs the Assembler to include the named file in the assembly. If no drive name is specified, the Assembler will begin searching for the file starting with drive A. If a drive is specified, the Assembler will begin the search with that drive. Filename extensions must be completely specified. Includes may not be nested. Each file included must have an END statement. The default number of drives is 2, but this value may be changed with the "DRIVES" directive. Below are some examples of the use of this directive.

```
INCLUDE   FILE.ASM
INCLUDE   A:FILE.ASM
INCLUDE   B:FILE.ASM
```

The first two examples are the same. The Assembler does not need to find the file on drive A; the drive name simply specifies which drive the Assembler should start searching from. The third example will start the search from drive B, so if the file is on drive A the Assembler will not find it, and will terminate the Assembly.

DRIVES EXP

This directive can be used to tell the Assembler how many drives are in the system. This is used only for included files and it enables the Assembler to make a clean exit (deleting all Assembler generated files) when it can't find the Include file. If used, this directive should appear before an "INCLUDE" directive. If only one drive exists and this directive is not used, a BDOS error will be generated by the operating system if the file is not found. The default number of drives is 2.

CONDITIONAL ASSEMBLY

IFZ EXP

The Assembler will assemble the statements following the directive up to an ELSE or ENDIF directive if the value of EXP is equal to zero. Conditional statements may be nested up to 248 levels.

IFNZ EXP
COND

Assemble the statements following the directive up to an ELSE or ENDIF directive if the value of EXP is not equal to zero. Conditional statements may be nested up to 248 levels.

IFTRUE EXP

This directive is actually the same as IFNZ, but is more logical when using assembly time comparisons. If the specified condition is true, then the following statements are assembled up to an ELSE or ENDIF directive. If the condition is not true, the statements up to an ELSE or ENDIF directive are not assembled.

IFFALSE EXP

This directive is the same as IFZ, and is the complement to IFTRUE. If the specified condition is false, then the following statements are assembled up to an ELSE or ENDIF directive. If the condition is true, then the statements up to an ELSE or ENDIF directive are not assembled.

IFDEF LABEL

This directive will activate a symbol table search, and if LABEL is found, then the statements following this one up to an ELSE or ENDIF directive will be assembled. If LABEL is not found, then the statements following this statement up to an ELSE or ENDIF directive will not be assembled.

IFNDEF LABEL

This directive is the complement of IFDEF. The symbol table is searched and if LABEL is not found, the statements following this one up to an ELSE or ENDIF directive are assembled. If LABEL is found, then the statements following this one up to an ELSE or ENDIF directive are not assembled.

IFSAME STRING1,STRING2

This directive compares STRING1 to STRING2, and conditionally assembles the statements following this statement depending on the result of the comparison. If the two strings are identical then the statements up to an ELSE or ENDIF directive are assembled. If the strings are not identical, then the statements up to an ELSE or ENDIF directive are not assembled. The strings may be one of two different types, namely with spaces or without spaces. However, both strings being compared must be of the same type. If the strings contain spaces, then the beginning and end of each string must be denoted with an apostrophe, with embedded apostrophes denoted by the use of two apostrophes. If the strings do not contain spaces, then the apostrophes are not required. This mode is very useful when comparing macro parameter arguments. In both cases, the strings must be separated with a comma. Below are some examples of the use of IFSAME.

```
IFSAME    'test string','test string'
IFSAME    '2500 A.D.''s','2500 A.D.''s'
IFSAME    X,Y
```

In the first example above, the strings contain spaces and therefore must be bracketed by apostrophes. The second example shows embedded apostrophes, which are represented by using two apostrophes. In the third example, a macro might be testing for a certain register, and since the strings do not contain spaces, they do not need to be enclosed in apostrophes.

IFDIFF STRING1,STRING2

This directive is the complement to IFSAME. If the two strings are not identical, the statements after this statement are assembled up to an ELSE or ENDIF directive. If the two strings are identical the statements up to an ELSE or ENDIF directive are not assembled. The syntax rules governing the form of the strings are the same as for IFSAME. See IFSAME for examples of the use of this directive.

IFEXT LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label has been declared external. An error message is generated if the label is not found.

IFNEXT LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label has not been declared external. An error messasge is generated if the label is not found.

IFABS LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label is absolute (i.e. not relocatable). External labels are considered to be relocatable. An error message is generated if the label is not found.

IFREL LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label is relocatable. External labels are considered to be relocatable. An error message is output if the label is not found

IFMA EXP

This directive is intended to be used inside a macro, and will scan the macro call line for the existence of the argument number specified by the value of EXP. If the argument exists, the statements following this one up to an ELSE or ENDIF will be assembled. If the argument does not exist, the statements following this one up to an ELSE or ENDIF will not be assembled. No arguments can be detected by having EXP = 0. In this case, if no arguments are present in the macro call line, the following statements are assembled, and if arguments are present in the macro call line, the following statements are not assembled. See the Macro section of this manual for examples of the use of this directive.

IFNMA EXP

This directive is the complement to IFMA, and checks the macro call line to see if the argument number given by the value of EXP exists. If the argument is not present, the statements following this one up to an ELSE or ENDIF are assembled. If the argument is present, the statements following this up to an ELSE or ENDIF are not assembled. The existence of any arguments at all can be detected by having EXP = 0. In this case, if there is at least one argument in the macro call line, the following statements will be assembled. If there are no arguments in the macro call line, the following statements will not be assembled. See the Macro section of this manual for examples of the use of this directive.

ELSE

Start of statements to be assembled if any of the above IF type of directives are false.

ENDC
ENDIF

Specifies the end of a conditional assembly block. When the Assembler detects unmatched IF - ENDIF pairs, an error message is output. Since recursive macros will almost always be controlled by IF type directives, the IFCLEAR directive may be needed. The difference between the two is that ENDIF is always executed, while IFCLEAR is not executed when it is inside a false conditional assembly block.

IFCLEAR

This directive performs exactly the same function as ENDIF, except that it is not executed when it is inside a false conditional assembly block. This directive can be used in a recursive macro to maintain balanced IF - ENDIF pairs, allowing the macro to eventually terminate, yet still taking advantage of the IF - ENDIF checking performed by the Assembler. This directive can be used to perform the same function when a macro contains a MACEXIT directive for early macro exits, since these would almost always be controlled by an IF directive of some sort. See the Macro section of this manual for examples of the use of this directive.

ASSEMBLY LISTING CONTROL

LIST ON

Turns listing on if directive listing was specified as the listing destination when the Assembler was first entered. This directive must always be used before LIST OFF. In other words, at the start of the program, LIST OFF is assumed.

LIST OFF

Turns listing off if directive listing was specified and LIST ON was executed. This is the default mode and therefore should only be used following a LIST ON directive.

MACLIST ON

Turns listing of MACRO expansions on. This is the default mode.

MACLIST OFF

Turns listing of MACRO expansions off. The default is on.

CONDLIST ON

Turns on listing of false conditional assembly blocks. This is the default mode.

CONDLIST OFF

Turns off listing of false conditional assembly blocks. The default is on.

PASS1 ON

Turns on the listing of pass 1. This can be used to help find errors due to the Assembler taking a different path on Pass 1 as compared to Pass 2. This condition will usually generate a 'Symbol value changed between passes' error. This directive can also be useful for finding nested conditional assembly errors.

PASS1 OFF

Turns off listing of pass 1 assuming PASS1 ON was executed.

PAGE
EJECT

Outputs a form feed to the listing device.

TITLE STRING
HEADING

Causes 'STRING' to be printed at the top of every page. If 'STRING' is not specified the 'TITLE' directive will be turned off. The title may be changed as often as desired and may be turned off at any time. The maximum title length is 80 characters. Also, the first two tabs between the TITLE directive and the start of the string, if they exist, will be ignored. All spaces and tabs after this will be included in the title.

SUBTITLE STRING

Causes 'STRING' to be printed at the top of every page. If 'TITLE' was executed, the subtitle will appear below it. If 'TITLE' was not executed or was turned off, the subtitle will still be output. If 'STRING' is not specified, the directive will be turned off. The subtitle may be changed as often as desired and may be turned off at any time. The maximum subtitle length is 80 characters. As with the TITLE directive, the first two tabs between the SUBTITLE directive and the start of STRING, if they exist, will be ignored and any spaces and tabs that appear after that will be included in the subtitle.

PW EXP

Sets the printer page width. The default page width is 132 columns.

PL EXP

Sets the printer page length. The default page length is 61 lines. The Assembler issues a form feed when this limit is reached or exceeded. If an error is encountered, the Assembler will output the form feed after the error message.

TOP EXP

This directive controls the number of lines form the top of the page to the page number. The default is zero (0).

ASSEMBLY TIME CALCULATIONS

The following list gives the allowed assembly time calculations. Also shown is their priority level. Priority level 7 operations are the first to be performed. Parenthesis may be used to force the calculations to proceed in a different order. All calculations can be performed using either 16 or 32 bit integer arithmetic with the exception of exponentiation which only uses an 8 bit exponent. The type of checks made on overflows depends on which mode the Assembler is operating in. See the next section for more information on this subject. The maximum number of pending operations is 16. Spaces are not allowed in between arguments.

OPERATION	PRIORITY	DESCRIPTION
Unary +	7	Optionally specifies a positive operand.
Unary -	7	Negates the following expression.
\ or .NOT.	7	Complements the following expression.
Unary >	7	Keeps the high order byte of the following address. This must be used to obtain relocatable byte address values.
Unary <	7	Keeps the low order byte of the following address. This must be used to obtain relocatable byte address values.
**	6	Unsigned exponentiation
*	5	Unsigned multiplication
/	5	Unsigned division
.MOD.	5	Remainder
.SHR.	5	Shift the preceeding expression right (with 0 fill) the number of times specified in the following expression.
.SHL.	5	Shift the preceeding expression left (with 0 fill) the number of times specified in the following expression.
+	4	Addition
-	4	Subtraction
& or .AND.	3	Logical AND
^ or .OR.	2	Logical OR
.XOR.	2	Logical exclusive OR

ASSEMBLY TIME COMPARISONS

The following list gives the assembly time comparisons which will return all 1's if the comparison is true and all 0's if the comparison is false:

=	or .EQ.	-	Equal
>	or .GT.	-	Greater than
<	or .LT.	-	Less than
.UGT.		-	Unsigned greater than
.ULT.		-	Unsigned less than

16 VERSUS 32 BIT ARITHMETIC

With the use of the MOD32 directive, the Assembler may be directed to perform arithmetic using either 16 or 32 bit arithmetic. The advantage of 32 bit arithmetic is that the chance of overflow is greatly reduced. Normally, overflow would only occur during an evaluation. In other words, the final result would 'fit' in 16 bits, but coming up with the result would cause an overflow. For example, the following instruction is part of the Assembler Source code, and is in a section that is used to automatically size all the Assembler buffers. The idea is that if a buffer does not meet certain requirements, an error message is to be output. This particular example deals with the Symbol Table buffer, which actually has it's own sub-buffer. The main requirement for the sub-buffer to perform properly is that it be an exact multiple of the main Symbol Table buffer. The following instruction will generate an error if this condition is not met.

```
.IFNZ  ((SYMLIM-SYMTBL+1).SHL.16)-(((SYMLIM-SYMTBL+1)/SYMBUFBS)
        *(SYMBUFBS.SHL.16))

  SYMBOL TABLE BUFFER NOT AN EXACT MULTIPLE OF SYMBOL TABLE SIZE.

.ENDIF
```

In this case, the statement in between the IF block is not an instruction, and so if the above calculation produces a non-zero result, the Assembler outputs an error because it does not recognize the instruction. You also get the statement itself, which becomes a 'customized' error message. The item of interest here is really the 16 bit left shifts, which are what make this technique work in the first place. Now, if the Assembler is running in 16 bit mode, these shifts obviously produce a zero result. But in 32 bit mode, such is not the case. What it does is essentially maintain a 16 bit remainder in all calculations, which happens to be exactly what is needed. So, 32 bit arithmetic can prove to be powerful.

While 32 bit arithmetic can be a plus, it can produce some strange results in two different cases. The first is for those users that are used to having Assembly Time Calculators turn over without question, such as the following example.

```
SIZE:           .EQUAL    1100H-CB00H
```

In this case, underflow actually exists, but it is probably an address calculation and it does not matter. For cases such as this, the 16 and 32 bit mode will provide two different results. The 32 bit mode will produce an error message indicating that the result is too large to fit in a 16 bit register, while the 16 bit mode will process this expression with no problem. The reason for this is that the 16 bit mode does not check for overflows, and the result therefore will always be at most a 16 bit result. On the other hand, the 32 bit mode is able to detect that the result is actually too large.

Another important item to keep in mind is that all values stored in the Symbol Table are 16 bit values. Therefore, when a symbol is involved in a calculation, it enters the calculation as a 16 bit number. Consider the following example where this creates a condition that appears to the Assembler to be a result that is larger than 16 bits, but there doesn't appear to be any reason for it.

```
FALSE:          .EQUAL      0000            ;Define FALSE
TRUE:           .EQUAL      .NOT.FALSE      ;Define TRUE

                .IFNZ       .NOT.FALSE
                MOV         A,R0
                .ENDIF

                .IFNZ       .NOT.TRUE
                MOV         A,R1
                .ENDIF
```

What happens here is the Assembler outputs a '# TOO LARGE' error for the .NOT.TRUE expression. This happens because TRUE was stored in the Symbol Table as FFFFH. The Assembly Time Calculator does not sign extend it, because it really doesn't know whether it is even a signed number or not. But when it takes the 32 bit ones complement, the result is FFFF0000H. Next the routine that was originally called by the IFNZ directive knows that it is to check for 16 bit results, and determines that the result will not fit in a 16 bit register. This would not occur if TRUE was generated in the expression. It is the fact that TRUE was passed in as a 16 bit value that causes the problem.

In summary, it is recommended that 32 bit arithmetic be used only for those cases where overflow is undesirable during the calculation, and then the Assembler should be put back into the 16 bit mode.

MACROS

DEFINITION

A macro is a sequence of source lines that will be substituted for a single source line. A macro must be defined before it is used. The Assembler will store the macro definition and, upon encountering the macro name, will substitute the previously defined source lines. Arguments may be included in the macro definition.

For macro definitions, dummy arguments may not contain spaces. However, for actual macro calls, arguments may be any type; direct, indirect, character string or register. Spaces are not allowed in arguments unless it is an Ascii string, in which case the string must be bracketed in apostrophes. If the string contains an apostrophe, this can be specified with two apostrophes in a row. Arguments will be passed through to any nested macros if the dummy argument names are identical. Macro nesting is limited only by the amount of disk space available.

To define a macro the ".MACRO" directive is used. A macro must have the ".MACEND" or ".ENDM" directive following the macro definition. The name of the macro is in the label field.

ARGUMENT SEPARATORS

In the macro call line arguments must be separated by commas, however leading spaces and tabs are ingnored. If no argument is present, a single comma will serve as a place holder. In a macro body, the following argument separators are allowed:

, + - * / ** \ & ^ = < > ( ) [ ] ;

.NOT. .AND. .OR. .XOR. .EQ. .GT. .LT. .UGT.

.ULT. .SHR. .SHL.

CONCATENATION

The broken bar character (¦ = hex 7C) is used as the string concatenation operator. Concatenation may only be performed inside of a macro.

MACRO EXAMPLES

The first example simply uses a macro to define an instruction that is not part of the standard 8748 instruction set.

```
CLRR0:          .MACRO
                MOV         R0,0
                .MACEND
```

This macro clears the register R0, and every time the Assembler finds this instruction, it will substitute the macro body for it, i.e. MOV R0,0.

Suppose instead of always clearing register R0, it is desired to clear any particular register. In this case, the string replacement facilities that macros provide can be used. Consider the following example.

```
CLEAR:          .MACRO      ARG1
                MOV         ARG1,0
                .MACEND
```

In this case, whenever the Assembler finds the instruction CLEAR in the mnemonic field, it will substitute the first argument in the operand field for the so called Dummy Parameter ARG1. To clear register R2 for example, the following source line would be used.

```
                CLEAR       R2
```

Here the Macro Processor would replace ARG1 with R2, and then pass this instruction to the Line Parser, which would assemble it as MOV R2,0.

This macro can be modified to have it clear either a register or a memory location by making use of conditional assembly directives. This could be done as follows.

```
CLEAR:          .MACRO      ARG1
                .IFSAME     A,ARG1          ;Register A ?
                CLR         A
                .ELSE
                .IFSAME     R0,ARG1         ;Register R0 ?
                MOV         R0,0            ;Clear it if so
                .ELSE
                .IFSAME     R1,ARG1         ;Register R1 ?
                MOV         R1,0            ;Clear it if so
                .ELSE
                MOV         R0,ARG1         ;Otherwise clear the
                MOV         @R0,0           ; memory location
                .ENDIF
                .ENDIF
                .ENDIF
                .MACEND
```

NESTED MACROS

Macros may be nested to any desired level, limited only by the size of the disk storage device. For the following example, assume that a macro that will clear the CPU registers A, R0 and R1 is wanted. This can be done by making use of CLEAR, the macro used for the above example.

```
CLR_REGS:       .MACRO
                CLEAR       A
                CLEAR       R0
                CLEAR       R1
                .MACEND
```

The Assembler would generate the following source code in place of this macro.

```
                CLR         A
                MOV         R0,0
                MOV         R1,0
```

Another way to write this macro is with multiple parameters, along with a check for which parameters are actually present using the IFMA directive.

```
CLEAR:          .MACRO      ARG1,ARG2,ARG3
                .IFMA       1                   ;1st argument present ?
                MOV         ARG1,0              ;Clear it if so
                .ENDIF
                .IFMA       2                   ;2nd argument present ?
                MOV         ARG2,0              ;Clear it if so
                .ENDIF
                .IFMA       3                   ;3rd argument present ?
                MOV         ARG3,0              ;Clear it if so
                .ENDIF
                .MACEND
```

This macro could be called using any combination of registers as shown below.

```
                CLEAR       A                   ;Clear register A
                CLEAR       A,R0                ;Clear registers A & R0
                CLEAR       R0                  ;Clear register R0 only
                CLEAR       A,R0,R1             ;Clear all three
```

LABELS IN MACROS

Labels are allowed in macro definitions. Labels may be defined in two ways: explicit or implicit. Explicit labels in the macro definition will not be altered by the Assembler. Implicit labels are followed by a '#'. The Assembler will substitute a 3 digit macro expansion number for the '#'. In this case, the label and the macro expansion number must not exceed 10 characters. A label that ends with a "#" symbol outside of a macro will receive the same expansion number as the last macro. This provides for renaming labels without having to know the actual macro expansion number. An argument may not be used to specify a label. If this is required, the argument may be equated to a label inside the macro. If the argument is a parameter that will be be changed inside the macro, it can be equated to a variable using the 'VAR' directive. See the section entitled Recursion for an example of this.

MNEMONIC REDEFINITION

The Assembler tables are searched in the following order:

1st - Assembler Directive Table
2nd - Macro Definition Table
3rd - 8748 Mnemonic Table

Therefore, 8748 mnemonics may be redefined using macros and Assembler Directives may be added but not redefined, as long as no decimal point preceeds them. This is due to the fact that any mnemonic field starting with a decimal point is automatically assumed to be a built in Assembler Directive.

RECURSION

When macros are nested, all the information required to return to the previous source code environment is saved on disk. This design is what enables the Assembler to store macros on disk and still allow nesting to any desired level. Recursion (a macro calling itself) is allowed and is achieved by detecting that the macro is already active, and inhibiting the writing of the previous source code environment. However, a macro may not call another macro that in turn calls the first macro. When this occurs, the original return information is destroyed and the Assembler will not be able to return to the previous environment. Furthermore, the Assembler is not able to detect this condition because when the second macro activates the first macro, which is already active, the Assembler assumes a recursive macro condition, and does not record the information needed to return to the second macro.

Below is an example of a recursive macro that reserves the number of data bytes defined by dummy argument ARG1 and fills them with the value specified by ARG2.

```
RESERVE:    .MACRO    ARG1,ARG2
COUNT:      .VAR      ARG1            ;Store # of bytes to save
            .IFZ      COUNT           ;Check for done
            .IFCLEAR                  ;Clear the pending IF
            .MACEXIT                  ; And get out
            .ENDIF
COUNT:      .VAR      COUNT-1         ;Else dec byte count
            .BYTE     ARG2            ;Reserve the byte
            RESERVE   COUNT,ARG2      ;Loop through again
            .MACEND
```

This macro would be called with a statement such as the following:

```
            RESERVE   10,55H          ;Reserve 10 bytes and
                                       store 55 hex in each one
```

It is perfectly legal for a recursive macro, such as the one in the above example, to call another recursive macro and so forth out to whatever level is desired.

ASSEMBLER ERROR MESSAGES

PROGRAMMING ERRORS

This section provides a list of the error messages output by the Assembler for reference along with a possible reason for the error message and an example of code that might produce the error. The reason it's not more definite is that some errors have a dependency on previous events.

```
Error   - SYNTAX ERROR
Meaning - Usually a missing comma or parenthesis.
Example - MOV   A,R0)                   ; Missing left parenthesis

Error   - CAN'T RESOLVE OPERAND
Meaning - Can't tell what the programmer intended.
Example - MOV   B,12H                   ; B isn't an 8748 register

Error   - ILLEGAL ADDRESSING MODE
Meaning - Can't address the operand using this form.
Example - RR    @R0                     ; Register Indirect illegal

Error   - ILLEGAL ARGUMENT
Meaning - Operand can't be used here.
Example - CLR   R0                      ; Only A is legal

Error   - MULTIPLY DEFINED SYMBOL
Meaning - Symbol defined previously (not inlcuding '.VAR')
Example - LABEL:    NOP                 ; 1st definition
          LABEL:    NOP                 ; 2nd definition

Error   - ILLEGAL MNEMONIC
Meaning - Mnemonic doesn't exist and wasn't defined as a Macro.
Example - TEST      R0                  ; There is no TEST instruction

Error   - # TOO LARGE
Meaning - The destination is too small for the operand.
Example - MOV       A,123H              ; 123H won't fit in 8 bits
Note    - This may occur when the Assembler is operating in 32
          bit arithmetic mode, even though the operand appears to
          be the correct size. This is due to the fact that
          although all internal operations are performed using 32
          bit arithmetic, the result is at most 16 bits. There-
          fore, when a previous result is involved in the calcu-
          lation, it is only a 16 bit number. See the "MOD32"
          directive for a further discussion of this subject.
```

```
Error   - HEX # AND SYMBOL ARE IDENTICAL
Meaning - A label exists that is exactly identical to a hex
          number that is being used as an operand.  Even the hex
          number indicator must be in the same place for this
          error to be generated.
Example - ABH:      NOP             ; Define label
                    MOV  A,ABH      ; Assembler can't tell between
                                    ; label and legal hex number.

Error   - UNDEFINED SYMBOL
Meaning - Symbol wasn't defined during pass 1.
Example - LABEL:    NOP             ; Label defined
                    JMP     LABLE   ; Typo makes it unrecognizable

Error   - RELATIVE JUMP TOO LARGE
Meaning - Destination address in a different page.
Example - LABEL:    .BLKB   130     ; Generate distance
          NEXT:     JZ      LABEL   ; LABEL in a different page
Note    - In an assembly that contains a large number of errors,
          this error should be inspected last because the program
          counter becomes out of sync.

Error   - ILLEGAL ASCII DESIGNATOR
Meaning - Bad punctuation on Ascii character.
Example - MOV       A,"T'           ; Must be enclosed by the same
                                    ; type, either " or '.

Error   - END OF OPERAND EXPECTED BUT NOT FOUND
Meaning - Usually a syntax or format error.
Note    - This error is the last check on any instruction before
          the Assembler proceeds to the next line and indicates
          that there are extra characters after a legal operand
          terminator.


Error   - ATTEMPTED DIVISION BY ZERO
Meaning - Divisor operand evaluated to 0.
Example - DATA:     .EQUAL      10/(8-(2**3))

Error   - TITLE/SUBTITLE EXCEEDS 80 CHARACTERS
Meaning - Buffers are sized for 80 byte titles and subtitles.
Example -  .TITLE '      ...  (81 bytes later)  ...   '

Error   - SYMBOL VALUE CHANGED BETWEEN PASSES
Meaning - Symbol value decode during pass 1 not = pass 2.
Example - LABEL:    .EQUAL    LABEL2    ; LABLE2 not defined yet
                                        ; (not including '.VAR')
Note    - This error is usually caused by the Assembler taking
          different paths on Pass 1 as compared to Pass 2 due to
          conditional directive arguments changing value.  The
          directive   PASS1   ON/OFF   can be useful in finding
          these types of errors.
```

```
Error   - NESTED CONDITIONAL ASSEMBLY UNBALANCE DETECTED
Meaning - Any '.IF' type instruction without a matching '.ENDIF'
Example -           .IFNZ     1          ; 1st IF
                    .IFNZ     1          ; 2nd IF
                    NOP                  ; intermediate code
                    .ENDIF               ; only 1 ENDIF
Note    - See  the Assembler Directive section for the use of the
          '.IFCLEAR' directive.

Error   - ILLEGAL EXTERNAL SYMBOL
Meaning - External reference can't be used here.
Example -           .EXTERNAL  LABEL     ; External declaration
                    JZ         LABEL     ; Relatives can't be ext.

Error   - PARAMETERS EXCEED 80 CHARACTERS
Meaning - The maximum Macro parameter length is 80 characters.
Example - TEST:     .MACRO  ARG1,ARG2 .. (80 chars later) .. ARG80

Error   - NESTED MACRO UNBALANCE DETECTED
Meaning - Extra .ENDM found during macro expansion.
Example -           MACRO1               ; Use macro
                    .ENDM                ; There is already i .ENDM
                                         ; in the macro definition.
```

SYSTEM ERRORS

The following errors are not really programming errors, but are detected by the Assembler and in some cases can help explain unexpected results, especially when used in conjunction with the PASS1 directive.

Error - MISSING END STATEMENT
Meaning - No .END found at end of file or end of included module.

Error - CAN'T FIND INCLUDE FILE
Meaning - Filename incorrect, search started from too high a drive number, .DRIVE directive should be executed or just simply a non-existent file.

Error - ILLEGAL NESTED INCLUDE
Meaning - One included file containing an .INCLUDE directive. This error may also indicate that an included file did not have an END statement.

Error - NOT ENOUGH PARAMETERS
Meaning - The Assembler was initiated with a command line without sufficient information. Especially check for listing destination and responses to the .ASK directive.

Error - OPEN FILE ERROR
Meaning - The Assembler cannot open the file. Usually denotes a non-existent file.

Error - CREATE NEW FILE ERROR
Meaning - The Assembler is not able to create the output file. This usually is due to a full disk. Note that because the Assembler creates 6 files, even for small programs this might occur. Also, when detected, the Assembler deletes all the files it has created up to that point, and so the disk may appear to have room. There must be at least 6 times the smallest allocated block size available.

Error - WRITE ERROR
Meaning - The most likely cause is a full disk. See also CREATE NEW FILE ERROR.

Error - CLOSE FILE ERROR
Meaning - This error indicates that the Assembler cannot close out one of the 6 files it creates, usually because of insufficient disk space or directory space. See also CREATE NEW FILE ERROR.

Error - RANDOM RECORD READ ERROR
Meaning - This is the type of error that will be detected when illegal operations involve data stored on the disk, such as when two macros are calling each other.

2500 A.D. 8748 CROSS ASSEMBLER LINKER

The 2500 A.D. Linker allows the user to write assembly language programs consisting of several modules. The Linker will resolve external references and perform address relocation for up to 128 files, with the only limit to the number of Global symbols, External symbols and the object code size being the amount of disk storage space available. The Linker will operate in either Prompt mode or Submit mode as described in the 'Linker Operating Instructions' section of this manual.

To match different output requirements, multiple versions of the Linker are provided. These different versions, identified by the names that they are shipped with, are described below.

LINK1 - Executable File Output

The Linker shipped with the name LINK1 should be used when the desired output is an executable object code file. This means that the code will be linked and relocated to the desired execution address and any gaps resulting from Origin directives will be filled in with zeroed bytes. Only Origin directives with ascending address values will be handled correctly. This is because of the origin gap filling that is required to obtain an executable file. Origin directives in descending order will probably generate an output file much larger than expected due to huge gaps being filled in. If Origin directives are in descending order and/or if Origin gaps are not to be filled in, LINK1 must not be used. The default output extension of LINK1 is 'TSK', chosen to imply an executable file, but not one that is executable on the host system.

LINK2 - Intel Hex File Output

The Linker shipped with the name LINK2 should be used when the desired output is an Intel Hex format file. This Linker will relocate and link all the input modules, but gaps resulting from Origin directives will not be filled in. Instead, a new Intel Hex record will be started. Therefore, Origin statements may be in descending order. The output file is not sorted in ascending address order, but appears exactly as the load map shows. The default output extension of LINK2 is 'HEX'.

LINKER OPERATING INSTRUCTIONS

In the descriptions which follow, it is assumed that the user has renamed the appropriate Linker to LINK.EXE for MSDOS and LINK.CMD for CPM 86.

PROMPT MODE

To run the Linker type: LINK

The program will respond with: INPUT FILENAME :

The user may enter a filename and an extension, or if just the filename is entered, the Linker will assume the extension is '.OBJ' since this is the default output extension from the 2500 A.D. Assembler.

After the user enters the filename, the program will ask for the Load Address. All addresses within the file will be relocated to this address. It is expected that the Load Address for the first file will be specified, but after that the user will normally want the files to start where the previous one left off, in other words 'stacked'. To stack files, just type a carriage return in response to the Load Address prompt.

For those cases where a file contains Global symbol values that another program needs, but the actual object code from that file is not needed (such as overlays) respond to the Load Address prompt with a minus sign (-) before the address. The Linker will use the Global symbol values but the object code from that file will not be included in the Linker output file. This type of file is referred to as a 'Reference Only' file, and appears that way in the Load Map. The Reference Only file will be completely relocated and linked, which means that the Linker can stack Reference Only type files. This is done by entering the '-' which signifies reference only, followed by a carriage return, which tells the Linker to stack the file on top of the previous file. See the section entitled 'Linker Examples' for examples of the use of this feature.

The Linker will continue to accept filenames and load addresses until the user types a carriage return in response to the FILENAME prompt. The filenames should be entered in the order in which they will be loaded into memory for execution.

After all input filenames have been entered, the Linker will prompt the user for the output filename. A carriage return only response will cause the Linker to name the output file with the same name as the first input file, but with an extension determined by the Linker being used. See the section entitled 'Linker Description' for the different Linker default output extensions.

SUBMIT MODE

When linking a large number of files, the Prompt mode is error prone. In this case the input to the Linker can be taken from a file. The file input to the Linker would contain all the answers to the prompts, just as though the Linker was being run in Prompt mode. The underbar (_) character may be used to create a 'carriage return only' type response in a file, resulting in a response that can be inspected by the user.

Below is an example of a file that would be used to run the Linker in Submit mode.

```
Input Filename
Load address
Input Filename
Load address (or optional '_' if stacking)
 ...
 ...
Last Filename
Last address (or optional '_' for stacking)
At least 1 space or '_' for no more filenames
Output Filename (or optional '_' for default followed by a <CR>)
```

Assuming the Linker prompt file was named LINK.DAT, the Linker would be run with the following command:

```
LINK      LINK.DAT
```

This command could either be issued from the console or included in a Batch file. Note that the carriage return after the Output Filename response is required.

Input filenames have the default extension of .OBJ , since this is the default output extension of the 2500 A.D. Assembler.

If a carriage return only response is used for the output filename, the Linker will use the first input filename along with the appropriate extension for the Linker being used as the output filename. These extensions are described in the section entitled 'Linker Description'.

LINKER OPERATION

This section describes some of the operations performed by the Linker. For a description of the different Linker operating modes see the section entitled 'Linker Examples'.

ASSEMBLY ORIGIN HANDLING

Normally, a program that was to be relocated would not have an .ORIGIN directive, and therefore would be assembled starting at address 0000. The Linker would then add the load address specified by the operator to the address calculated by the Assembler to obtain the relocated address. However, the original assembly address does not have to be 0. Therefore, if the program was assembled at an address other than 0000, the result that can be expected is the assembly address plus the load offset address. The Load Map generated by the Linker always shows the value that the program has been linked and relocated to run at. Care must be exercised when stacking modules with other modules that make use of descending Origin statements. In this case, the user must insure that the Linker has a definite address on which to stack the modules. This can be accomplished by using absolute load addresses for the modules that have descending Origins and then stacking only those modules that have ascending addresses. LINK1 can never be used for files that have descending Origins.

CODE & DATA ADDRESS SPECIFICATIONS

Code and Data addresses are specified as Code first followed by Data, with a comma separating the two. If no Data address is specified, then all Data sections will be relocated and stacked on top of the Code sections. The Code address may be skipped by the use of just a comma. If no Code address is specified but a Data address is, all the Code sections will be relocated and stacked on top of the Data sections. Code and Data may be processed independently by explicitly responding to the Load Address prompt for each section.

GLOBAL - EXTERNAL SYMBOLS

The Linker matches external symbols (if any) with global (or public) symbols. If multiple global symbols are encountered, an error message is output. The load offset value is added to the assembly value of the global symbol before using it as an operand. If the external symbol is a 4 or 8 bit value, and the global symbol is too large an error message is output with the symbol name, the name of the file the symbol was declared GLOBAL in and the name of the file the symbol was declared EXTERNAL in. If an external symbol has no matching global symbol, an error message is output along with the name of the file that the symbol was declared EXTERNAL in.

ADDRESS RELOCATION

Addresses are relocated by adding the load address to the offset decoded by the 2500 A.D. Assembler. Normally the program would be assembled starting at location 0000, but it doesn't have to be. The load address will simply be added to any address generated by the Assembler.

The Assembler maintains a table of attributes associated with each symbol used in the program. If the label simply precedes an instruction, then it is tagged as relocatable. If the label is defined in an ".EQUAL" directive, then the relocatability of it depends on the operand field type. If the operand contains no relocatable tokens, then the expression is not relocatable. If the operand contains only one relocatable token, then the expression is relocatable. If the operand contains two or more relocatable tokens, then the expression is not relocatable.

Byte values are only relocatable candidates if the unary greater than '>' sign is used for the high byte and/or the unary less than '<' sign is used for the low byte. These operands are subject to the same relocation rules as full 16 bit address values.

Following are some examples illustrating these points.

```
LABEL1:   NOP              ;The label is defined to be equal
                           ; to the address of an instruction
                           ; and therefore is relocatable.

LABEL2:   .EQUAL   LABEL1  ;The label is defined to equal a
                           ; value that was tagged as relo-
                           ; catable. Therefore, LABEL2 is
                           ; also relocatable.

LABEL3:   .EQUAL   10      ;The label is defined to equal a
                           ; constant. Therefore, LABEL3 is
                           ; not relocatable.

LABEL4:   .EQUAL   $+10    ;The label is defined to equal a
                           ; relocatable value plus a non-
                           ; relocatable value. Since only
                           ; one value is relocatable, the
                           ; symbol LABEL4 is relocatable.

LABEL5:   .EQUAL   10+$    ;The label is defined to equal a
                           ; non-relocatable value plus a re-
                           ; locatable value. Since only one
                           ; value is relocatable, LABEL5 is
                           ; relocatable.
```

```
LABEL6:    .EQUAL     LABEL5-LABEL2
                                  ;The  label is defined to  equal  a
                                  ; relocatable  value minus  another
                                  ; relocatable  value,  producing  a
                                  ; non-relocatable result.
```

The last example is worth remembering when using the Assembler to do things such as calculate data sizes. Consider the following example of a table of data values, with the number of bytes being calculated automatically at assembly time by the Assembler, allowing the programmer to add or delete from the table without having to remember to change the data block size.

```
DATA:              .BYTE      0
                   .WORD      10
                   .BYTE      20
                   .BLKB      5

DATA_SIZE:         .EQUAL     $-DATA
```

The Assembler will calculate the size of the data block, and because the result is not relocatable, the Linker will not alter the data block size.

LINKER EXAMPLES

This section consists of examples intended to demonstrate the use of the Linker. The Linker prompts will be shown in upper case, and the user responses will be shown in lower case. The <cr> symbol denotes a carriage return and is shown only when no other response to a prompt is desired. Otherwise, all inputs are assumed to be terminated with a carriage return. Also, for the purpose of these examples, forget about the fact that filenames must be limited to 8 characters.

Single File Assembled At Desired Run Address

The first example is the case of just one file which has been assembled error free at the desired run address by the use of the ORIGIN directive. Since only the Code section address is input, if the file contains Data sections they will be stacked on top of all the Code sections.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : 0
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

The above will cause the Linker to read a file by the name of 'FILENAME.OBJ' , add 0 to all Code section relocation addresses, add the total size of all the Code sections to all Data section relocation addresses, and output a file with the same name as the input filename and a default extension as described in the section entitled 'Linker Description'.

If the Data section load address is specified but the Code section load address is omitted, then all the Code sections will be stacked on top of the Data sections. This is specified as shown below.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : ,0
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

If both the Code and Data sections were assembled at the desired address by the use of the ORIGIN directive, then an offset of 0 must be specified for both sections as shown below.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : 0,0
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

Single File Assembled At 0000

The following example is the same as the above example except that the program either did not contain an ORIGIN directive or the value or the ORIGIN operand was 0000.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : 100
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

This would cause the Linker to read a file by the name of FILENAME.OBJ, relocate all relocatable Code section values to 0100 Hex, achieved by adding 100 Hex to each address offset generated by the Assembler, and then relocate all relocatable Data section values by adding 100 Hex plus the total size of the Code sections to them. The result is the Data sections stacked on top of the Code sections. The output would be a file with the same name as the input file and with an extension which depends on the Linker used. The possible extensions are described in the section entitled 'Linker Description'.

The following example is the same as the above example except that the Data sections would be relocated first, and the Code sections would be stacked on top of all the Data sections.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : ,100
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

The example below shows the case where both the Code sections and Data sections have been given addresses to be relocated to.

```
INPUT  FILENAME : filename     LOAD ADDRESS (OFFSET) : 100,200
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

Note that if the Code sections extend beyond the Data sections starting address, LINK1 cannot handle this situation because it is impossible to generate an executable object file since both Code and Data are residing at the same address. The resulting file will more than likely grow larger than 64K as LINK1 attempts to fill in the gap, at which time an error message advising the user of this situation will be output.

Two Files Assembled at Run Address, Stacked

This example consists of two files that have been assembled error free at the desired run address, each one containing references to the other. The first file contains the entry point to the program. What is desired is to link the two files together just as if they had been assembled as one file, with the Code sections being relocated and stacked on top of each other first, followed by the Data sections being relocated and stacked on top of all the Code sections.

```
INPUT  FILENAME : filename1    LOAD ADDRESS (OFFSET) : 0
INPUT  FILENAME : filename2    LOAD ADDRESS (OFFSET) : <cr>
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

In this case, the Linker would read a file by the name of FILENAME1.OBJ, calculate the size of the object code and add the value 0 to the initial ORIGIN value. Then the Linker would read a file by the name of FILENAME2.OBJ and add the value 0 + the initial ORIGIN value in FILENAME1.OBJ + the size of the object code in FILENAME1.OBJ. Note that this will only produce stacked code if FILENAME2 does not have an ORIGIN statement in it, or if it does, that it's value is 0. The output from the Linker would be a file with the same name as the first input filename, but with an extension as determined by the Linker used. See the section enttitled 'Linker Description' for the extensions output by the different Linkers.

The following example is the same as the above example except that the Data sections are to be stacked on top of each other first, then all the Code sections are to be stacked on top of the entire Data section.

```
INPUT  FILENAME : filename1    LOAD ADDRESS (OFFSET) : ,0
INPUT  FILENAME : filename2    LOAD ADDRESS (OFFSET) : <cr>
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

In the following example both the Code and Data starting addresses are specified, and Code is stacked on Code and Data is stacked on Data.

```
INPUT  FILENAME : filename1    LOAD ADDRESS (OFFSET) : 100,500
INPUT  FILENAME : filename2    LOAD ADDRESS (OFFSET) : <cr>
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

Two Files Assembled At 0000, Stacked

This example consists of two files that are to be linked together into one file. They may or may not reference each other. Both files were assembled at location 0, and therefore must be relocated to the desired run address.

```
INPUT  FILENAME : filename1     LOAD ADDRESS (OFFSET) : 100
INPUT  FILENAME : filename2     LOAD ADDRESS (OFFSET) : <cr>
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

The Linker will read and relocate the object code from the file FILENAME1.OBJ, then read and relocate the object code from the second file FILENAME2.OBJ and relocate it so that it starts after the last byte of the object code from FILENAME1.OBJ. Any references to the other file will be resolved during this process. The result will be one file with all addresses relocated to run at 0100 Hex, with all Code sections first, followed by any Data sections.

Single File With Multiple Origin Statements

This example considers the case where a file contains more than one ORIGIN directive. The load address will be added to whatever value was specified in the source file.

```
INPUT  FILENAME ? : filename     LOAD ADDRESS (OFFSET) ? : 100
INPUT  FILENAME ? : <cr>

OUTPUT FILENAME ? : <cr>
```

The Linker will read a file by the name of FILENAME.OBJ and begin adding 100 Hex to the relocatable addresses. When it detects that an ORIGIN statement was executed the action taken by the Linker depends on the Linker being used. If LINK1 is being used it will check for an Origin gap. If there is one, it will fill it with zeroed bytes, since this is required to generate an executable object file. Therefore, if the Origins are in descending order, LINK1 cannot correctly generate an executable file since it will fill the gap by rolling over past FFFF Hex to 0000 Hex. This will more than likely produce a file larger than 64K and the Linker will output an error message advising the user of this condition. Versions of the Linker that do not fill in the Origin gaps, such as LINK2 which outputs an Intel Hex file, can be used with descending Origins.

Two Files, One Used Only For Global Values

This example illustrates the concept of Reference Only files. Assume the first file contains object code assembled at 0 and also contains external references to say, a jump table of system routines. The jump table already exists and is not to be made part of the program. It is simply the jump addresses that are neeeded. Assume the jump table resides at address A000 Hex.

```
INPUT  FILENAME : filename1     LOAD ADDRESS (OFFSET) : 100
INPUT  FILENAME : filename2     LOAD ADDRESS (OFFSET) : -A000
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

The Linker would read a file called FILENAME1.OBJ and link all extenal references to a file called FILENAME2.OBJ . Before the addresses from FILENAME2.OBJ were used, A000 Hex would be added to them, therefore that file would probably have been assembled at 0. If not, then the desired result could have been obtained by entering -0 for the OFFSET for FILENAME2.OBJ . The output file would consist of the object code from FILENAME1.OBJ with all references to FILENAME2.OBJ resolved, but no object code from FILENAME2.OBJ would be included. Note that the first file should contain the program entry point (assumed equal to the first ORIGIN statement operand if no entry point is specified after the END directive, or assumed to be a relocatable 0000 if no ORIGIN directive was executed) and therefore the Reference Only file was specified after the main object code file.

In the above example, the Data sections followed the same attributes as the Code sections, i.e. if the Code section was Reference Only, so was the Data section. Both Code and Data sections may be specified independently, so that Code may be Reference Only, but not Data, or Data may be Reference Only, but not Code.

Three Files, Two Stacked & Used Only For Global Values

This example is similar to the above example with the addition of another Reference Only file which is to be stacked on top of the first Reference Only file.

```
INPUT  FILENAME : filename1    LOAD ADDRESS (OFFSET) : 100
INPUT  FILENAME : filename2    LOAD ADDRESS (OFFSET) : -A000
INPUT  FILENAME : filename3    LOAD ADDRESS (OFFSET) : -<cr>
INPUT  FILENAME : <cr>

OUTPUT FILENAME : <cr>
```

This example illustrates the ability of the Linker to stack Reference Only files. The Linker will read a file with the name of FILENAME1.OBJ, relocate it by adding 100 Hex to all relocable values and then resolve external references to FILENAME2.OBJ and FILENAME3.OBJ . Before any external references are used, the object code size of FILENAME2.OBJ will be calculated, and then all relocatable addresses used by FILENAME3.OBJ will be relocated to produce the equivalent of FILENAME3.OBJ stacked on top of FILENAME2.OBJ . The result will be a file with the same name as the first input file, and with an extension as determined by the Linker used, with all external addresses relocated to a file that would be the equivalent of FILENAME2.OBJ and FILENAME3.OBJ stacked together. The output however will consist of only the file FILENAME1.OBJ . As many Reference Files as desired may be included in the link (as long as the 128 file limit is not exceeded) and may be stacked, calculated with separate load addresses or both. The same holds true for the files that are to be included in the output file. They may be stacked, specified with separate load addresses or both.

In the above example, the Data sections followed the same attributes as the Code sections, i.e. if the Code section was Reference Only, so was the Data section. Both Code and Data sections may be specified independently, so that Code may be Reference Only, but not Data, or Data may be Reference Only, but not Code.

TSK FILE TO INTEL HEX FORMAT CONVERTER

The 2500 A.D. TSK File to Intel Hex Format Converter will convert an executable object code file to the Intel hex format. It is intended to be used in conjunction with files output by LINK1 (See the section enttitled 'Linker Description' for an explanation of LINK1). The file output by HEX may be much larger than the actual program itself. This is due to the fact that the smallest input file size is one record, i.e. 128 bytes. This then gets converted to twice the size, resulting in 256 bytes. Since additional information like start characters, load addresses and checksums are also included, the smallest file output from this converter is 3 records long. If exact sizes are required, LINK2 should be used.

To run the Converter type : HEX

The Converter will respond with the following prompt:

ENTER INPUT FILENAME :

If no input filename extension is given, the Converter will default to TSK. The output file will have the same name as the input filename, but with an extension of HEX.

After the user enters the input filename, the Converter will ask for the starting address. This is the address that the code actually runs at, and is coded into the output file.

The Intel Hex Format is described below.

Record Mark Field - This field signifies the start of a record, and consists of an Ascii colon (:).

Record Length Field - This field consists of two Ascii characters which indicate the number of data bytes in this record. The characters are the result of converting the number of data bytes in binary to two Ascii characters, high digit first. An end of file record contains two Ascii zeros in this field. The maximim number of data bytes in a record is 255. This converter uses 32 data bytes per record.

Load Address Field - This field consists of the four Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of address.
Low digit of high byte of address.
High digit of low byte of address.
Low digit of low byte of address.

In an end of file record, this field consists of either four Ascii zeros, or the program entry address. Since the Converter does not know the program entry address, this field will always be zeros in an end of file record.

Record Type Field - This field identifies the record type, which is either 00 for data records or 01 for an end of file record. It consists of two Ascii characters, with the high digit of the record type first, followed by the low digit of the record type.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in the end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field, the record type field and the data field. This sum is then negated (2's complement) and converted to two Ascii characters, high digit first.

LOWER TO UPPER CASE SOURCE CODE CONVERTER

The 2500 A.D. Lower to Upper Case Source Code Converter will convert all characters in a file from lower to upper case except for strings enclosed in apostrophes. If the string contains an apostrophe, this can be specified with two apostrophes in a row. This rule is consistent with most assemblers.

To run the Converter type : CASE

The Converter will respond with the following prompt.

ENTER SOURCE CODE INPUT FILENAME :

After the user enters the input filename the Converter will display the output filename prompt as shown below.

ENTER SOURCE CODE OUTPUT FILENAME :

After this, the Converter will display each converted line at the terminal.